The FYSOS File System (version 1.62) (dated: 14 March 2021)

by Benjamin David Lunt

This is a small attempt at a file system. This file system will not break any new records, but at least it is something that may help you with your file system creation. I merely created this file system to test my virtual file system code within FYSOS. I wanted to make sure that there was no file system dependant code within the virtual file system code.

A brief summary

The FYSOS file system (FYSFS) contains a root directory and a FAT similar to the MS® FAT file system. However, the FYSFS differs in that the directory entry and the FAT entry(s) are stored in the same place. No need to read multiple locations just to get the directory and the fat entry of a file.

The FYSFS allows for a filename of unlimited chars in length (though the host may limit it to 255 chars), a 64-bit file size, and 64-bit FAT entries. This allows for very large files and volume sizes.

An overlook of a typical FYSOS FS partition

There are a few items in a FYSFS volume that must be set to fixed values. The boot sector and its location as well as the size and location of the **SuperBlock**.

The **Boot Sector** is always at **L**inear **S**ector **N**umber zero, or **LSN** 0. The FYSFS leaves the first 16 sectors free for the boot. These 16 sectors are reserved for the boot code, if needed. These 16 sectors should contain all of the necessary code and data needed to load the operating system, or at least a loader file to do so.

There is a **SuperBlock** at LSN 16 that contains information about the volume. However, we have a chicken-and-the-egg situation here. We don't know the Base LBA where our volume is located until we read the **SuperBlock**. The Master Boot Record (MBR) knew where to read, since it read from the base it has stored in the Partition Table. However, there is no documented standard way to pass this base to the volume's boot sector. We need to know the base to read the **SuperBlock**, but need to read the **SuperBlock** to know the base. Therefore, we need to have the Base LBA in the first sector of the booted code, since the BIOS has read that sector for us.

To do this, the last 14 bytes of the first sector of the Boot Sector must have the following format. Remember that this is at offset (512 - 14) no matter the sector size.

```
  bit32u sig;           // Random serial number for volume
  bit64u base_lba;      // 64-bit base lba this sector occupies
  bit16u boot_sig;      // 0xAA55  (0x55 0xAA)
```

The **sig** field is a randomly calculated volume serial number used to help identify which volume was booted from. The **base_lba** field will be here and in the **SuperBlock**, to make it easier on the driver code. The **boot_sig** field is the standard boot signature value for all bootable devices. All FYSOS file systems must have this 14-byte block at this address in the first sector of the volume.

The remaining 15 sectors are used for any remaining code and data that is needed to load the operating system. These sectors should not be assumed for anything else. If you use these sectors for other data, be aware that they may be overwritten at any time by an updated version of the boot sector code.

The SuperBlock

LSN 16 contains the **SuperBlock** and has the format shown on the next page.

```
struct S_FYSFS_SUPER {
  bit32u sig[2];        // signature   'FYSF' 'SUPR' (0x46595346 0x53555052)
  bit16u ver;           // version in BCD (0x0161 = 1.61)
  bit16u sect_clust;    // sectors per cluster (1 or a power of 2 < 512)
  bit8u  encryption;    // if non-zero, type of encryption used
  bit8u  bitmaps;       // number of bitmaps (1 or 2)
  bit16u bitmap_flag;   // flags describing the bitmaps
  bit32u root_entries;  // count of 'slots' in the root (128 -> 65532 && (mod 4 = 0))
  bit64u base_lba;      // physical sector LSN 0 occupies (also in boot sector)
  bit64u root;          // LSN pointer to root.  Must be in data block
  bit64u data;          // LSN pointer to the first sector of the data block
  bit64u data_sectors;  // count of sectors in data block
  bit64u sectors;       // total sectors in volume
  bit64u bitmap;        // LSN of list of bitmap sectors
  bit64u bitmapspare;   // LSN of list of bitmap sectors (second copy)
  bit32u chkdsk;        // Seconds since 0:00 1-1-1980 when last chkdsk was ran
  bit32u lastopt;       // Seconds since 0:00 1-1-1980 when last optimized
  bit32u flags;         // volume flags
  bit32u crc;           // crc of this (and the copy) super block
  GUID   guid;          // serial number in GUID format (see below)
  char   vol_label[64]; // asciiz volume label
  bit8u  filler[250];   // filler (to pad to 512 bytes)
  bit8u  enc_check[90]; // if encryption is used, this is a check.  See the section
};                      //   on Encryption for more on this field.
```

All LSN pointers in the **SuperBlock** are zero based from the start of the volume. All LSN pointers within the **data block** are zero based from the start of the **data block**. i.e.: all starting sectors and fat entries for any file within the **data block** are zero based within that **data block**, not the start of the volume. Typically, the **data block** occupies all remaining sectors, except for the bitmaps, explained below. All values are written and read in little-endian format, with the least significant byte read/written first and the most significant byte last.

The **flags** field in the **SuperBlock** only has bits 1:0 used while the remaining 30 bits must remain reserved and preserved. As with any reserved fields, any value read from a reserved field must be written back as the same value. Bit 0 is use to indicate if the volume's filenames are case sensitive. If set, they are case sensitive. This flag should only be modified at format time. It can be set to use case sensitivity at any time during the life of the volume, however it **must not** be cleared after a volume has been used unless only after a complete check of the volume to make sure there are no conflicts with case sensitivity, only then can it be cleared.

Bit 1 in the **flags** field indicates if there is a copy of this **SuperBlock** at the end of the volume. If there is, this copy must be in the first sector of the last **cluster** of the **data block**. Note that it is in the last cluster, not last sector. This cluster must also be marked as 'system' in the bitmap(s). To verify that this copy is a valid copy, you must perform a CRC check on the copy. Only if it passes can you copy it to LSN 16. If the primary **SuperBlock** ever goes bad, you can always check the last cluster of the volume for the copy. If it passes the CRC check and has the correct signature, it is safe to assume that it is a valid copy.

The CRC check follows that of the official CRC-32 standard, and is coded similar to the code found at the following URL:

http://gnu-darwin.org/www001/src/ports/emulators/mupen64-glide/work/glide64-0.7.SP8/CRC.cpp

The GUID serial number follows the rules explained at the following URL:

http://en.wikipedia.org/wiki/Globally_unique_identifier

Please note that all values are written as little-endian instead of big-endian as the URL above mentions. All values in this specification are written as little-endian including the GUID. The calculation algorithm used for this serial number is not specified within this specification. Any values may be used.

From this point on, the rest of the items on the volume are not located in fixed locations. Each item is pointed to by the **SuperBlock**.

The Bitmaps

On a typical FYSFS partition, the bitmap(s) would be next. These bitmaps are double-bit representations of the use of each **cluster** in the partition. This two-bit field must be one of the following four values:

```
    00b = unoccupied.  Is free and can be used for cluster allocation.
    01b = occupied.  Is an occupied cluster, and should be left as so.
    10b = deleted.  Was occupied, now is free for use.  See notes below.
    11b = system.  Used to mark bad and/or system non-movable clusters.
```

So that files can be undeleted, you can mark a cluster as free for use using 10b, marking them as deleted. Then when checking for a free cluster, you can skip any cluster that is marked as deleted. Therefore, you can undelete a file later. However, when unoccupied clusters are becoming scarce, the optimizer should be run to change some/all deleted entries to unoccupied entries, including any deleted slots in the corresponding directories. It is not of this specification to specify when to change and how many deleted entries to unoccupied entries.

The high two bits (bits 7:6) in the first byte of the bitmap represents **Cluster 0**, while the low two bits (bits 1:0) represents **Cluster 3**. The high two bits in the second byte of the bitmap represents **Cluster 4**, and so on. There are typically **two** bitmaps, though the **SuperBlock** contains a field stating how many there are. There can be 1 or 2, no more, no less. Bits 1:0 in the **bitmap_flag** field in the **SuperBlock** denote which bitmap is to be used and if the inactive is to be updated as the active bitmap is updated.

Within the **SuperBlock**, if bit 0 of the **bitmap_flag** field is clear, the first bitmap is the active bitmap. If set and a second bitmap is present, then it is active. If this bit is set and the **bitmaps** entry is 1, then the OS should return an error at time of mount.

If bit 1 is clear, the active bitmap is used, while the inactive bitmap is not touched. If this bit is set, the inactive bitmap needs to be updated to match the active bitmap at a time when the system is in a stable state. The time and state to update the inactive bitmap is to be determined by the host. However, no more than 128 modifications should be made to the active bitmap before the inactive bitmap is updated. A modification in this sense is a single write to the bitmap. If multiple bits will be modified, but there is only one read/write sequence, this is considered a single modification. No more than 128 of these modifications should be made before the second bitmap is updated to match the active bitmap.

The Data Area

On a typical FYSFS partition, the **Data Area** follows the bitmap(s) and contains the **Root Directory** within that **Data Area**. The **Root Directory** block can be anywhere within this **Data Area**. The **root_entrys** field in the **SuperBlock** determines the size of the **Root Directory**. The Root Directory must be continuous and not fragmented. The Root Directory can be resized as long as it remains continuous on the disk and the root_entrys field is updated. Each entry, or slot, is 128 bytes fixed and there are six (6) types of data blocks that can occupy each slot. The bitmap(s) must be marked ‘used’ for the clusters that occupy any data in the **data block**, which includes directory blocks.

The first type of slot is the actual directory entry of the file:

```
struct S_FYSFS_ROOT {
  bit32u sig;            // signature of this slot type
  bit32u attribute;      // file attributes
  bit8u  resv[5];        //
  bit8u  fat_entries;    // fat entries in this slot
  bit8u  crc;            // crc of this slot.
  bit8u  scratch;        // OS scratch byte.  Should be zero when written to disk.
  bit32u created;        // seconds since 00:00:00 1-1-80
  bit32u lastaccess;     // seconds since 00:00:00 1-1-80  (read or written)
  bit64u fsize;          // file size
  bit32u fat_continue;   // next entry that continues the FAT (0 if none)
  bit32u name_continue;  // next entry that continues the name (0 if none)
  bit16u flags;          // file flags (encryption, etc)
  bit8u  namelen;        // length of name in this slot
  bit8u  resv1[5];       //
  bit8u  name_fat[80];  // name/fat buffer of this entry
};
```

A definition of the fields:

sig	set to ‘SLOT’ (0x534C4F54) to denote the first slot in a chain.
attribute	the attributes of the file. (see next page)
fat_entries	the count of FAT entries in **this** slot.
crc	lower 8 bits of 32-bit crc explained previously.
scratch	OS may use this byte as it likes when in memory, however, it should be written as zeros to the disk, though is not mandatory. (See note below)
created	Seconds since 00:00:00 1 Jan 1980.
lastaccess	Seconds since 00:00:00 1 Jan 1980, when last read or written.
fsize	the file size in bytes.
fat_continue	the number of the next slot that contains more FAT entries for this filename. Set to zero if no more slots are needed.
name_continue	the number of the next slot that contains more chars of the filename. Set to zero if no more slots are needed.
flags	file flags used by the OS
namelen	Length of the filename in **this** slot.
name_fat[80]	remaining 80 bytes of the slot used for the filename and/or FAT entries. See below for more information on what is in this field.

A directory entry (**slot**) can contain both the name and/or the FAT for a specified file. Since the FYSFS allows a filename to be many chars in length, a slot must contain a link to the next slot that contains the remainder of the name if the name will not fit in the current slot. The same goes for the FAT.

The name and the FAT entries are located in the **name_fat** field. The name must be first and must have at least 1 char included in the first slot. The name must also be end padded with nulls to the next 32-bit alignment. However, the name does not have to end in a null if the last char is the byte before a 32-bit aligned byte. The **namelen** field does not include any null padding in its length.

If the name occupies the whole **name_fat** field and needs more room, the host must allocate another unused slot, place this zero-based slot number in the **name_continue** field and then continue placing the name in the new slot (not allowing slot 0 to be a continuation slot). See below for the format of this new slot. The 32-bit FAT entries follow the same format as the name does. The only difference is that there can not be any FAT entries before any filename chars in the same slot. All FAT entries must proceed filename chars. To know where the first FAT entry is located in the **name_fat** field, you simply skip **namelen** bytes and start on the next dword aligned entry. If the **namelen** field contains a value of 77 or greater, then the FAT entries start at the slot pointed to by the **fat_continue** field.

Please note that when calculating the offset of the first FAT entry, you can not assume that **name_fat** will be dword aligned in memory. When you calculate using

```
            dword_offset = ((x + 3) & ~0x03)
```

make sure that the offset of **name_fat** is not included in the calculation above. If I were going to use the C language for my driver, my source would be similar to:

```
    ptr = (bit32u *) (root[i].name_fat + ((root[i].namelen + 3) & ~0x03));
```

Notice that the offset of **name_fat** was not included in the rounding calculation.

The first FAT entry in the slot chain is the starting cluster number of the file. With this in mind, there must always be at least one FAT entry, whether in this slot or a fat_continue slot, unless this is a zero sized file. The cluster number is zero based from the start of the **data block** given in the **super_block**.

The starting slot does not have to contain any FAT entries as long as there is at least one fat_continue slot.

Since the starting slot (a slot with a **sig** of 'SLOT') may use only 32-bit fat entries, this slot may only contain fat entries if the file's clusters are 2^32 from the start of the volumes **data block**. However, this sets a limit on the volume size used. If you use a cluster size of 128 sectors, with 512-byte sectors, this still gives you a volume size of 128 * 512 * 2^32 bytes, or a 256-terabyte volume. However, with drives starting to exceed this size, and/or with smaller cluster sizes, you may want to have a volume larger than this limit. To do so, simply place all FAT entries in a fat_continue slot, set the **large** flag, and use 64-bit FAT entries. This has a max volume size limit of 512 * 512 * 2^64, or 2^82, or a 4.8 *yottabyte* volume size. A *yotta* is a 1 with 24 digits after it, or a terabyte of terabytes.

There is a Catch22 when it comes to using 64-bit FAT entries in a continuation slot. When creating a continuation slot, you must check all of the entries to make sure that they are 32-bit entries, or you must use the **large** flag. Therefore, you must check 28 cluster entries to see if they will fit in the given space in this slot. Now, let's say the 15th entry is the first 64-bit entry, you would now need to use the **large** flag to store the entries. By doing so, you only have enough room to store the first 14 entries as 64-bit entries, and now all entries in this continuation slot are now only 32-bit entries and no longer need the **large** flag. Yet, when you clear the large flag, you now include the 15th entry again. Since either method, in this situation, would use the same number of slots, it is up to your driver what to do in this situation.

However, please note that if one FAT entry is 64-bit, then every other one will most likely be that way too.

The valid **attributes** of a file are:

```
  #define FYSFAT_ATTR_ARCHIVE   0x00000001
  #define FYSFAT_ATTR_SUB_DIR   0x00000002
  #define FYSFAT_ATTR_SYSTEM    0x00000008
  #define FYSFAT_ATTR_HIDDEN    0x00000010
  #define FYSFAT_ATTR_READ_ONLY 0x00000020
```

Other bits in this field are reserved for future use.

The **created** field contains the seconds since 00:00:00 1-1-1980 to the date this file was created. The **lastaccess** field contains the seconds since 00:00:00 1-1-1980 when the file was last read or written. When a file is first created, these two fields should be identical. If this entry points to a sub-directory, the **lastaccess** field must be ignored, though not assumed zero.

The **fsize** field is the size of the file in bytes. If this entry points to a sub-directory, the **fsize** field must be ignored, though not assumed zero.

The **scratch** field may be used by the driver as a scratch byte and it should be written to the disk as zeros, though it is not mandatory. However, due to multiple hosts using this partition, this field, if read from the disk as non-zero, should not be assumed valid. FYSOS uses bit 0 in this field to indicate if this slot has been modified before the commit to disk has taken place. The continue slots defined below also have this field. **This field must remain at offset 15 in all types of slots**.

The valid **flags** of a file are:

```
 Bits 15:3 -- These bits are reserved and preserved.
 Bits  2:0 -- Type of filename used:
            000b = Ascii
            001b = UTF-8
            010b = UTF-16
  011b -> 111b = reserved
```

The **filename** can be stored using one of the types specified above. All characters in the first slot and any continuation slot for this filename, and this filename only, use this type of character storage. If type 000b is used, please remember that it is not stored as ASCIIZ, meaning it may not have a terminating null char. It is recommended that once the encoding type is chosen for a filename, that it remains this type. A host may change the encoding type of a filename, though it is not recommended.

The **crc** field contains the lower 8 bits of the 32-bit CRC check (explain ion a previous section) in this 128-byte slot not counting this field. i.e.: zero this byte, then do a 32-bit CRC of all 128 bytes, then store the lower 8-bits of this CRC value to this byte. **This field must remain at offset 14 in all types of slots**.

Continuation Slots

The second and third types of slots are continuation slots. The format of the **name_continue** and **fat_continue** slot is as follows:

```
struct S_FYSFS_CONT {
  bit32u sig;           // 'NAME' for name, 'FAT ' for FAT
  bit32u previous;      // pointer to the previous slot in this slot chain
  bit32u continue;      // next slot that continues the remainder (0 if none)
  bit8u  count;         // length of name or count of FAT entries in this slot
  bit8u  flags;         // see below
  bit8u  crc;           // crc of this slot
  bit8u  scratch;       // scratch byte
  bit8u  name_fat[112]; //
};
```

A definition of the fields:

sig	set to 'NAME' (0x4E414D45) for a **name_continue** slot
	set to 'FAT ' (0x46415420) for a **fat_continue** slot
previous	points to the previous slot number in this slot chain.
next	points to the next slot number in this slot chain. zero if no more.
count	is the number of chars in **this** slot not counting the null char
or	
	is the count of FAT entries in **this** slot
flags	name_continue:
	this field is ignored and should be zero.
	fat_continue:
	only bit 0 is used. Bits 7:1 should remain zero.
	Bit 0: if not zero, the fat entries in **this** slot are 64-bit.
crc	lower 8 bits of 32-bit crc explained above
scratch	see the previous description for this field.
name_fat	contains the chars for the name, or up to 28 32-bit or 14 64-bit FAT entries.

The fourth type of a 128-byte slot is the **empty** slot. It simply contains zeros in all fields if a truly empty slot. The fifth type contains 'DLTD' (0x444C5444) in the **sig** field denoting that this slot chain has been deleted. When deleting slot chains, you must mark all slots in that slot chain with this id. The last type is the **sub-directory** type and contains 'SUB ' (0x53554220). See the section on Directories for more information on this type.

If the **sig** field contains anything other than one of these five types, or the sub-directory type explained below, it is considered a used slot. With this version of the specification, this is a bad slot and will be deleted with the next "Check Disk" or "Optimize" performed. Since a later version of this specification may contain other valid slot **sig** values, you should not modify any slot that does not have the six defined values above. Typically, all empty slots should have a value of zero in the **sig** field.

The following rules apply when deleting or adding slots to a slot chain:
- You may move the current FAT entries further down the slot to make room for a longer name as long as there is room in this slot to do so. If there is not room to move all FAT entries down, you can do one of two things:
  - you can create a new slot for the remaining chars in the name
    or
  - you can "scroll" the existing FAT entries through to the next existing fat_continue slot(s), creating a new slot if necessary.

- You may make room in this slot by moving all or some of the FAT entries to another new (or occupied by this slot chain) slot. You may use multiple partial slots, but this would fragment the slot chain and use unnecessary space. Typically, you would never have more than two partial slots in each slot chain, one partial for the name and one for the FAT. If the host operating system supports the optional optimization of FYSFS systems, the user can optimize all fragmented/partial slots to have at most two partials.
- A slot may not contain FAT entries that do not start directly on the next dword aligned location after the filename.
- When the OS calls to rename the filename and the new name is shorter than the current name, you must move the FAT entries in this slot to align after the new shorter name. You can leave "blanks" at the end of the slot if so desired. i.e.: you do not have to append the newly moved FAT entries with FAT entries from an existing fat continue slot.
- The first slot in the slot chain is the only slot in this slot chain that can have both name and FAT entries. Every other slot in the slot chain **must** contain a valid NAME or FAT slot header and contain only that type of item.
- When deleting a file, marking all associated slots with 'DLTD', you must not change any other part of the slot, especially the CRC. This way, when you go to undelete the slot, you change the signature to 'SLOT' and check the CRC. If it matches, you have correctly undeleted this slot. If the CRC does not match, you continue to change the signature to one of the remaining slot types until the CRC matches. This is the only way to tell what type of slot it was before the deletion.

When allocating a new slot for a new file, you must mark the **sig** entry as 0x534C4F54. This denotes the first slot of this slot chain. When allocating a new slot to continue the name and/or FAT, you must mark the **sig** entry as 0x4E414D45 for continuing the name, and 0x46415420 for continuing the FAT.

Remember that the **namelen** field contains the length of the name in the current slot only, not the total length of the name used by the whole slot chain. Same goes for the **fat_entries** field. It contains the count of fat entries in this slot only.

Directories

FYSFS directories are implemented similar to the FAT12/16 FS. A directory entry (slot) simply points to a data block containing another set of directory entries (slots). A sub-directory can grow just like a file does. It is up to the host to decide on how many clusters to allocation on sub-directory creation, however at least one (1) must be allocated at creation time. When the host requests a new **slot** to be created in this sub-directory and there is no more room, at least one more cluster can be allocated and added to the fat entry list of the SUB slot. In other words, the sub-directory can grow to use as much of the volume as desired. See the note below on the size limit of a directory.

The first slot in the **Root** directory **must** be a starting slot, a **sig** value of 'SLOT'. This must be so, since zero is used to denote that there are no more continue_slots in slot chain. When implementing this FS, a "find empty slot" routine should ignore the first slot in any directory block when looking for an empty slot when creating a continue_slot for a slot chain.

When a SLOT entry has the attribute of FYSFAT_ATTR_SUB_DIR, meaning that this slot points to a sub-directory, this slot must contain **only** 1 fat entry and this fat entry must point to the starting cluster of the sub-directory's directory block. This slot does not contain the fat entries for the sub-directory. The 'SUB ' slot type in the sub-directory's first slot contains the fat entries (see below). As with normal file slot entries, the parent slot may contain as many NAME_CONT slots as needed, but may contain only 1 FAT_CONT slot if one is needed due to the name being longer than 77 chars, and/or the cluster number is a 64-bit cluster number.

When a sub-directory is created, the first slot **must** be filled with a sub-directory slot entry. This entry has the format shown on the next page.

```
struct S_FYSFS_SUB {
  bit32u sig;           // signature of this slot type (0x53554220)
  bit64u parent;        // starting cluster of parent directory
  bit8u  resv;          //
  bit8u  fat_entries;   // fat entries in this slot
  bit8u  crc;           // crc of this slot.
  bit8u  scratch;       // OS scratch byte.  Should be zero when written to disk.
  bit32u created;       // seconds since 00:00:00 1-1-80
  bit32u resv1;         //
  bit64u fsize;         // file size (size of directory block in bytes)
  bit32u fat_continue;  // next entry that continues the FAT (0 if none)
  bit32u slot;          // slot index in parent's directory of this slot entry
  bit16u resv2;         //
  bit8u  namelen;       // must be zero
  bit8u  resv3[5];      //
  bit32u fat[20];       // fat buffer of this entry
};
```

A definition of the fields:

sig	set to 'SUB ' (0x53554220) to denote the sub directory.
parent	starting cluster of parent directory. (Super->Root == root dir)
resv	must be zeros
fat_entries	the count of FAT entries in **this** slot.
crc	lower 8 bits of 32-bit crc explained above
scratch	OS may use this byte as it likes when in memory, however, it should be written as zeros
created	Seconds since 00:00:00 1 Jan 1980.
resv1	must be zeros
fsize	the size of this directory block in bytes
fat_continue	the number of the next slot that contains more FAT entries for this directory block. Zero if no more slots are needed
slot	slot index in parent's directory block of this slot entry's 'SLOT'
resv2	must be zeros
namelen	this must remain zero since there are no chars for the name
resv3	must be zeros
fat[20]	room for twenty 32-bit cluster numbers

This slot type indicates to the system, how many and where the remaining clusters are for this sub-directory. It also gives the starting cluster for the parent directory block. If more than 20 clusters are needed, or 64-bit cluster numbers are needed, you can use a standard CONT slot as you would a regular SLOT entry. This CONT slot is not required to be directly after the 'SUB ' slot entry. It can be anywhere within this sub-directory's slot entries.

Please remember that this slot type **must** be the first slot in the directory block. Also, a note about the size of a sub directory. A sub-directory block cannot grow beyond 2^32 * 128 and can only have 2^32 slots. However, this allows a sub-directory's slots to occupy one gigabyte of the drive, and have four billion directory entries. Please note that this limit is only the limit for the slots, not the files themselves.

The **root directory** may grow as long as it does not exceed the limit above **and** it uses consecutive clusters. To grow the root directory, you may use unused consecutive clusters at the end of the current root directory or you may copy the current root directory to another location in the volume, then marking the old location as free. You may also move clusters at the end of the root directory to other locations updating the corresponding slot chains to make room for a larger root directory. If you move the root directory to a larger consecutive block of clusters, do remember to update the super block on the volume.

With this type of sub-directory allocation, I could change the way the root directory is stored. It could be exactly as detailed above with sub-directories. However, this would give more work to the boot code. I may change it later, though I believe I will leave it as is to not add more work for the boot code.

Note that all slots in a directory must remain within that directory block. You cannot have a continuation slot chain span directory blocks. Any continuation slot chain must remain within the directory block allocated to that directory.

Please note that if the directory block uses FAT_CONT slots to store the clusters of the directory block, you must be careful with their placement in the directory block. For example, if you use only 64-bit cluster numbers, requiring a FAT_CONT slot, this continuation slot **must** be within the first cluster of the directory block since you have not read any other clusters yet. Also, if you use more continuation slots, for the sub-directory block, that can fit in the first cluster, you must have one of the entries in the first few slot's point to the remaining clusters. Do not put a continuation slot in a position where you have not made access to that cluster yet.

For an example of a sub-directory, let's say you have a slot in the root directory block that has the name 'test_dir' and the FYSFAT_ATTR_SUB_DIR attribute set. This slot must only have 1 FAT entry, no matter the size of the sub-directory it points to, and must be the cluster number of the first cluster of the sub-directory's directory block.

Then, the first slot in the sub-directory's directory block must be the 'SUB ' slot type explained above. It will contain all of the cluster numbers that occupy the directory block including the first cluster, which was in the root directory's slot above. The 'SUB ' slot type here can use standard FAT_CONT slot, if needed, to allocate all of the clusters used. This slot also contains the cluster number of the first cluster of the parent's directory block, along with the slot offset of the 'test_dir' entry above. This is so that a sub_directory can find its parent if needed.

A note on the Slot Number

A slot number, used in determination of the next continuation slot, is zero based from the start of this directory block. For example, if in the root directory, a continuation slot has a continuation slot number of 5, this points to the sixth slot from the start of the root directory. With a 128-byte slot, this points to offset 640 (5 * 128). When within a sub-directory, the slot number is zero based from the start of the SUB slot. Therefore, all slots are zero based from the start of the directory block they reside in. As described earlier, this gives a limit of 2^32 count of slots per directory block.

Notes on optimizing/defragmenting the volume

A typical optimized volume would have as much of the name in the first slot ('SLOT') as possible, with any needed name_continue slots ('NAME') following this slot, and with any fat_continue slots ('FAT ') following the name_continue slots. Since a decent host, on opening the file, will load and cache the whole FAT chain in memory, the FAT chain will be read and then written only once per opening/closing of the file, file sharing aside. However, the name of the file may be needed multiple times when looking for filenames, due to many find_first/find_next function calls.

Defragmentation of the volume is typical of most volume types. Simply place all file clusters in consecutive placement on the volume.

When the bitmap(s) get scarce of unoccupied entries (00b), you may change deleted entries to unoccupied entries to free up disk space. However, please remember to also change any corresponding DELETED slots to FREE slots. The best way to do this is to find DELETED slots in a directory, change them to FREE slots, then change all corresponding bitmap entries for this slot chain, moving to the next slot chain as needed.

Sector sizes

Since each slot is 128 bytes, as long as the sector size is evenly divisible by 128, it doesn't matter what the sector size is. Usually it is 512, but this may change in the future to 1024, 4096, or even more. If there ever is a sector size found that is not evenly divisible by 128, the slots occupy the first bytes of the sector up to the point where another 128-byte slot will not fit. This lost space is just that, lost. A 128-byte slot **must not** cross a sector boundary. However, it is very unlikely that a sector size will not be evenly divisible by 128.

Encryption

Encryption on a FYSFS volume is done on the cluster level. Encryption is used on a volume if the 'encryption' field in the **SuperBlock** is non-zero. The non-zero value indicates what type of encryption is used. As of this version of this specification, only the value of 0x01 is allowed and indicates RC4 encryption explained at http://www.fysnet.net/cypher.htm and http://en.wikipedia.org/wiki/RC4.

Since it would be very destructive to use an invalid key on a volume that first didn't verify the encryption key, the **SuperBlock** contains a field at the end of the block for verification. The volume does not store the key used for encryption, this would defeat the purpose. However, the volume does store the 10-byte suffix for the key.

When the volume is mounted, the user should be asked for the encryption key. Once received, the system should verify the key is valid by using the 10-byte suffix, stored in the **SuperBlock**, and the given key to decipher the enc_check[] field in the **SuperBlock**.   This field is as follows:

```
    Suffix:  The 10-byte suffix used to encrypt the data
 GUID guid:  Copy of the volume's serial number
 vol_label:  Copy of the volume's 64-byte label
```

To verify the given key is correct, use the Suffix field appended to the given key to decipher the guid and vol_label fields above. If they correctly match the guid and vol_label fields in the **SuperBlock**, then you have a valid key.

Please note that all encryption is at the cluster level and is only encrypted or decrypted a cluster at a time. Only clusters in the **Data Area** are encrypted. The bitmaps, boot block, and SuperBlock are not to be encrypted. The encryption/decryption should be done as the cluster is read or written.

Since encryption can and may be restrictive in most countries, please verify all laws before passing encryption files across country lines.

Please note that the CRC field in the **SuperBlock** is calculated on the encrypted data in the enc_check[] field as it was read from the disk. Do not calculate the CRC from the unencrypted data in this field when writing back to disk.

Here are a few examples of a typical FYSFS root directory.

First is an example of a slot that needs no continuation.
i.e.: The name and the FAT entries fit in this slot.

slot 0: (128 bytes)

```
00000000  54 4F 4C 53 01 00 00 00-00 00 00 00 00 01 33 00     TOLS............
00000010  78 56 34 12 12 34 56 78-EF CD AB 89 67 45 34 12     ................
00000020  00 00 00 00 00 00 00 00-00 00 11 00 00 00 00 00     ................
00000030  52 65 61 64 20 6D 65 20-66 69 72 73 74 2E 74 78     Read.me.first.tx
00000040  74 00 00 00 55 AA 55 AA-00 00 00 00 00 00 00 00     t...U.U.........
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
```

offset	**contents**	**description**
00000000	0x534C4F54	the New Slot id: 'SLOT'
00000004	0x00000001	attribute of the file: FYSFAT_ATTR_ARCHIVE
00000008	0x0000000000	reserved
0000000D	0x01	number of 32-bit FAT entries in this slot
0000000E	0x33	crc of this slot
0000000F	0x00	scratch
00000010	0x12345678	file creation date: seconds since 00:00 1-1-80
00000014	0x78563412	file last closed date: seconds since 00:00 1-1-80
00000018	0x0123456789ABCDEF	File size
00000020	0x00000000	FAT continue slot. 0 = no continue
00000024	0x00000000	Name continue slot. 0 = no continue
00000028	0x0000	File flags: none

```
0000002A  0x11        Name length:  Length in chars of the file name in this slot.
0000002B  0x0000000000 reserved
00000030  name        The filename starts here.  It does not have to end in null
                       unless we need padding to next dword aligned position.
00000044  0xAA55AA55  The first FAT entry.  This is the cluster number of the
                       starting cluster of the file.
```

remaining bytes are zeros since there is not enough name chars and/or fat entries to fill the slot.

Second is an example of a slot that needs name continuation due to a renaming of the filename to a length that no longer fits in the slot. This example chose to select a new slot for the continuation of the name instead of “scrolling” the FAT entries through to the next slot or to a new slot.

slot 0: (128 bytes)

```
00000000  54 4F 4C 53 01 00 00 00-00 00 00 00 00 0E 33 00     TOLS............
00000010  78 56 34 12 12 34 56 78-EF CD AB 89 67 45 23 01     ................
00000020  00 00 00 00 01 00 00 00-00 00 18 00 00 00 00 00     ................
00000030  54 68 69 73 20 69 73 20-61 20 76 65 72 79 20 6C     This.is.a.very.l
00000040  61 72 67 65 20 66 69 6C-55 AA 55 AA 55 AA 55 AB     arge.filU.U.U.U.
00000050  55 AA 55 AC 55 AA 55 AD-55 AA 55 AE 55 AA 55 AF     U.U.U.U.U.U.U.U.
00000060  55 AA 55 B0 55 AA 55 B1-55 AA 55 B2 55 AA 55 B3     U.U.U.U.U.U.U.U.
00000070  55 AA 55 B4 55 AA 55 B5-55 AA 55 B6 55 AA 55 B7     U.U.U.U.U.U.U.U.
```

offset contents description

```
00000000  0x534C4F54  the New Slot id: ‘SLOT’
00000004  0x00000001  attribute of the file: FYSFAT_ATTR_ARCHIVE
00000008  0x0000000000 reserved
0000000D  0x0E        number of 32-bit FAT entries in this slot
0000000E  0x33        crc of this slot
0000000F  0x00        scratch
00000010  0x12345678  file creation date: seconds since 00:00 1-1-80
00000014  0x78563412  file last closed date: seconds since 00:00 1-1-80
00000018  0x0123456789ABCDEF  File size
00000020  0x00000000  FAT continue slot.  If 0 = no continue
00000024  0x00000001  Name continue slot.  If 0 = no continue
00000028  0x0000      File flags:  none
0000002A  0x18        Name length:  Length in chars of the file name in this slot.
0000002B  0x0000000000 reserved
00000030  name        The filename starts here.  It occupies 24 bytes.
00000048  FAT         The FAT Entries occupy the remainder of the slot.
```

Now for the name continuation slot:

slot 1: (128 bytes)

```
00000000  45 4D 41 4E 00 00 00 00-00 00 00 00 09 00 33 00     EMAN............
00000010  65 6E 61 6D 65 2E 74 78-74 00 00 00 00 00 00 00     ename.txt.......
00000020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
```

offset contents description

```
00000000  0x4E414D45  the Continue Name id: ‘NAME’
00000004  0x00000000  The previous slot number (parent)
00000008  0x00000000  The next slot number (0 = not used)
0000000C  0x09        number of chars in this slot
0000000D  0x00        flags
0000000E  0x33        crc
```

```
0000000F  0x00        scratch
00000010              continuation of the filename
```

An example of a FAT continuation slot chain would be identical to the above, except that a few of the id's would be changed, and slot 1 would contain FAT entries. The only difference in **slot 0**, other than the name modification (for example purposes), is the two continuation fields. The **name_continue** field is now zero, and the **fat_continue** field is now one. No other field was changed.

slot 0: (128 bytes)
```
00000000  54 4F 4C 53 01 00 00 00-00 00 00 00 00 0E 33 00     TOLS............
00000010  78 56 34 12 12 34 56 78-EF CD AB 89 67 45 23 01     ................
00000020  01 00 00 00 00 00 00 00-00 00 18 00 00 00 00 00     ................
00000030  54 68 69 73 20 69 73 20-61 20 76 65 72 79 20 6C     This.is.a.very.l
00000040  61 72 67 65 20 66 69 6C-55 AA 55 AA 55 AA 55 AB     arge.filU.U.U.U.
00000050  55 AA 55 AC 55 AA 55 AD-55 AA 55 AE 55 AA 55 AF     U.U.U.U.U.U.U.U.
00000060  55 AA 55 B0 55 AA 55 B1-55 AA 55 B2 55 AA 55 B3     U.U.U.U.U.U.U.U.
00000070  55 AA 55 B4 55 AA 55 B5-55 AA 55 B6 55 AA 55 B7     U.U.U.U.U.U.U.U.
```

offset contents description
```
00000000  0x534C4F54  the New Slot id: 'SLOT'
00000004  0x00000001  attribute of the file: FYSFAT_ATTR_ARCHIVE
00000008  0x0000000000 reserved
0000000D  0x0E        number of 32-bit FAT entries in this slot
0000000E  0x33        crc of this slot
0000000F  0x00        scratch
00000010  0x12345678  file creation date: seconds since 00:00 1-1-80
00000014  0x78563412  file last closed date: seconds since 00:00 1-1-80
00000018  0x0123456789ABCDEF  File size
00000020  0x00000001  FAT continue slot.  0 = no continue
00000024  0x00000000  Name continue slot.  0 = no continue
00000028  0x0000      File flags:  none
0000002A  0x18        Name length:  Length in chars of the file name in this slot.
0000002B  0x0000000000 reserved
00000030  name        The filename starts here.  It occupies 24 bytes.
00000048  FAT         The FAT Entries occupy the remainder of the slot.
```

And for the FAT continuation slot:

slot 1: (128 bytes)
```
00000000  20 54 41 46 00 00 00 00-00 00 00 00 03 00 33 00     .TAF............
00000010  55 AA 55 B3 55 AA 55 B4-55 AA 55 B5 00 00 00 00     U.U.U.U.U.U.....
00000020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
```

offset contents description
```
00000000  0x46415420  the Continue Name id: 'FAT '
00000004  0x00000000  The previous slot number (parent)
00000008  0x00000000  The next slot number (0 = not used)
0000000C  0x03        number of FATS in this slot
0000000D  0x00        bit 0: 32-bit or 64-bit entries?
0000000E  0x33        crc
0000000F  0x00        scratch
00000010              continuation of the FAT entries
```

Next is an example of a sub-directory slot. It contains the name and a single FAT entry pointing to the cluster of the first slot in the sub-directory.

slot 0: (128 bytes)

```
00000000  54 4F 4C 53 02 00 00 00-00 00 00 00 00 01 33 00     TOLS............
00000010  78 56 34 12 12 34 56 78-EF CD AB 89 67 45 23 01     ................
00000020  01 00 00 00 00 00 00 00-00 00 08 00 00 00 00 00     ................
00000030  74 65 73 74 5F 73 75 62-55 AA 55 AA 00 00 00 00     test_subU.U.....
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
```

offset contents description

```
00000000  0x534C4F54  the New Slot id: 'SLOT'
00000004  0x00000002  attribute of the file: FYSFS_ATTR_SUB_DIR
00000008  0x0000000000 reserved
0000000D  0x01        number of 32-bit FAT entries in this slot
0000000E  0x33        crc of this slot
0000000F  0x00        scratch
00000010  0x12345678  file creation date: seconds since 00:00 1-1-80
00000014  0x78563412  file last closed date: seconds since 00:00 1-1-80
00000018  0x0123456789ABCDEF  File size
00000020  0x00000000  FAT continue slot.  0 = no continue
00000024  0x00000000  Name continue slot.  0 = no continue
00000028  0x0000      File flags:  none
0000002A  0x08        Name length:  Length in chars of the file name in this slot.
0000002B  0x0000000000 reserved
00000030  name        The filename starts here.  It occupies 24 bytes.
00000038  FAT         The FAT Entries occupy the remainder of the slot.
```

Then the cluster pointed to by the single FAT entry above contains the 'SUB ' type slot described below.

slot 0: (128 bytes)

```
00014800  20 42 55 53 00 00 00 00-00 00 00 00 0A 33 00     .BUS.........._.
00014810  C4 13 91 AD 00 00 00 00-00 A0 00 00 00 00 00 00     ................
00014820  00 00 00 00 02 00 00 00-00 00 00 00 00 00 00 00     ................
00014830  08 00 00 00 09 00 00 00-0A 00 00 00 0B 00 00 00     ................
00014840  0C 00 00 00 0D 00 00 00-0E 00 00 00 0F 00 00 00     ................
00014850  10 00 00 00 11 00 00 00-00 00 00 00 00 00 00 00     ................
00014860  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
00014870  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00     ................
```

offset contents description

```
00000000  0x534C4F54  the New Slot id: 'SUB '
00000004  0x0000000000000000 parent (cluster number of the root)
0000000C  0x00        reserved
0000000D  0x0A        number of 32-bit FAT entries in this slot
0000000E  0x33        crc of this slot
0000000F  0x00        scratch
00000010  0x12345678  file creation date: seconds since 00:00 1-1-80
00000014  0x00000000  reserved
00000018  0x0123456789ABCDEF  File size (0x0A * clusters_per_sect * bytes_per_sect)
00000020  0x00000000  FAT continue slot.  0 = no continue
00000024  0x00000002  slot of parent entry
00000028  0x0000      reserved
0000002A  0x00        Name length (must be zero)
0000002B  0x0000000000 reserved
00000030  FAT         The FAT Entries occupy the remainder of the slot.
```

Please note that all of the CRC's in the examples above are not correct. I used the value 0x33 just to show where it would go.

Revisions

14 Mar 2021
 - Fixed grammar errors, typos, and added clarifications. No structural modifications were made.

20 Oct 2018
 - Updated the signature at the end of the first sector of the volume.

03 Apr 2012
 - minor typo and grammar corrections.

24 Mar 2012
 - swapped the location of the SUB->parent and ->resv fields to help alignment.
 - added to the SLOT->flags field to indicate the type of encoding for the filename.
 - added notes about a parent's fsize and modification SLOT fields

21 Mar 2012
 - Rearranged the members of a Sub Directory slot a little.

20 Mar 2012
 - Changed the way Directories are stored on the disk.
 - This removed a few items from the SLOT type structure and added a new slot type.
 - As with most recent changes, this breaks any existing code/volumes. However, since no one has been complaining, I don't see a problem. If someone complained, that would mean that someone other than myself is using this FS. :-)

18 Mar 2012
 - Changed the bitmaps to use 2 bits per cluster.
 - Added information on encryption.

5 Mar 2012
 - Removed the BPB and placed any required fields into the Super Block.

24 Feb 2012
 - Minor document additions/changes.
 - Removed the DriveNum field of the BPB. This value is passed in DL from the boot code.

07 Dec 2009
 - Removed the bytes_per_sector field of the BPB. The host should already know the sector size in bytes.
 - Added the notes about sector sizes.

14 May 2009
 - Minor documentation updates to match current specs.

16 June 2008
 - Minor grammar changes and additions.
 - A few more details
 - Increased sectors per cluster limit to 512

15 June 2008
 - Changed the sectors field in the BPB to reserved.

14 June 2008
 - Moved the crc fields in the slots to byte 14, so that they will be in the same place in all types of slots.

7 June 2008
- Added the case sensitive information.
- moved the scratch byte in the SLOT's to be the same position no matter the slot
  type. This way it doesn't matter the slot type, one can read/write the scratch
  register at will. Makes it much better for the "dirty" flag. Unfortunately this
  breaks any existing images. But, as far as I know, I am the only one using this FS anyway. (smile)

5 June 2008
- Added the scratch field to the slot(s) structure.
- Specified that the BPB->Serial number should be a 32-bit value of seconds since
  00:00:00am on 1 Jan 1980.
- just a few small typo's, grammar issues.

7 Oct 2006
- made S_FYSFS_BPB.base_lba 64-bit, and made .resv1 a byte instead of 5 bytes.

1 Oct 2006
- removed five of the six reserved dwords in the Root Slot field. This
  gives room for 20 more chars in the name, or 5 more fat entries in the
  first slot
- changed the large field in the continuation slot to use bit 0 instead of
  the whole byte.
- the root_entrys field in the bpb now allows 128 entires as the minimum.
- the bpb.drivenum field needs to be set to the DL value given by the BIOS
- added information about the Volume attribute usage.
- other modifications to the text for clarity and detail.

8 April 2006
- removed "descriptor" from the BPB and combined the reserved fields.
  However, this did not move or change the structure of the BPB.
- added details on optimizing/defragmenting the volume.

7 April 2006
- Added more details and cleared up a few description to the specs.
- No format changes were made to the specs.

19 June 2005
- Moved the bitmaps field from the BPB to the SuperBlock
- Added the data_sectors field in the SuperBlock
- Changed the size of the bitmap flags
- Added a few small details

24 June 2005
- Added the copy_valid field in the BPB and a little more
  info about the validity of this copy.

25 Sept 2005
- Added a few comments to the documentation. No actual format
  of the file system was modified.

27 Sept 2005
- Added documentation about directory's and modified a field of
  the Root Directory slot structure.

9 Oct 2005
- changed the size of some of the entries in the super to 64-bit entries.
- added a field in a fat continue slot to denote whether the FAT entries are
  32-bit or 64-bit.